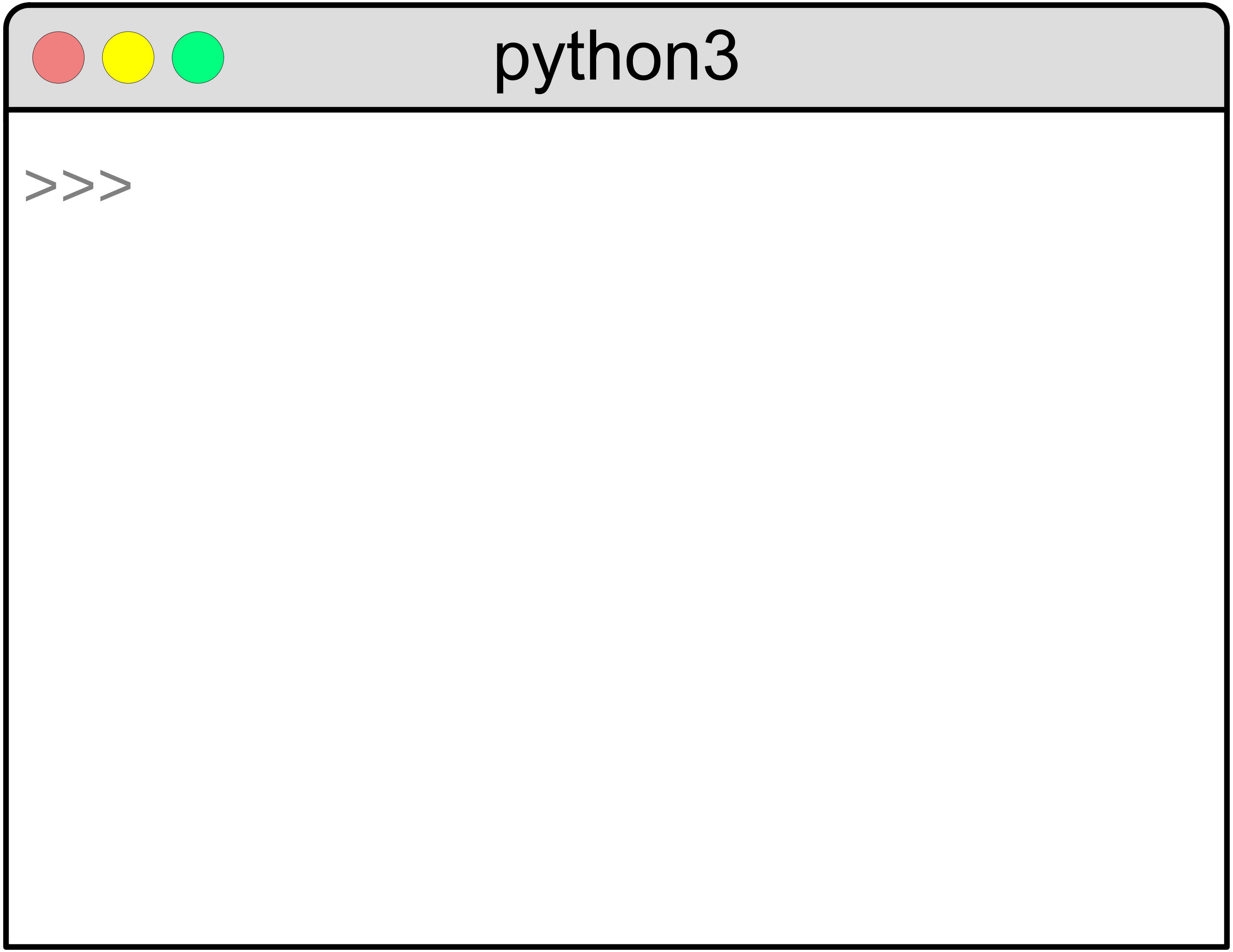
# Context-Free Grammars

# A Motivating Question


python3
>>>

python3

```
>>> (137 + 42) - 2 * 3
```

python3

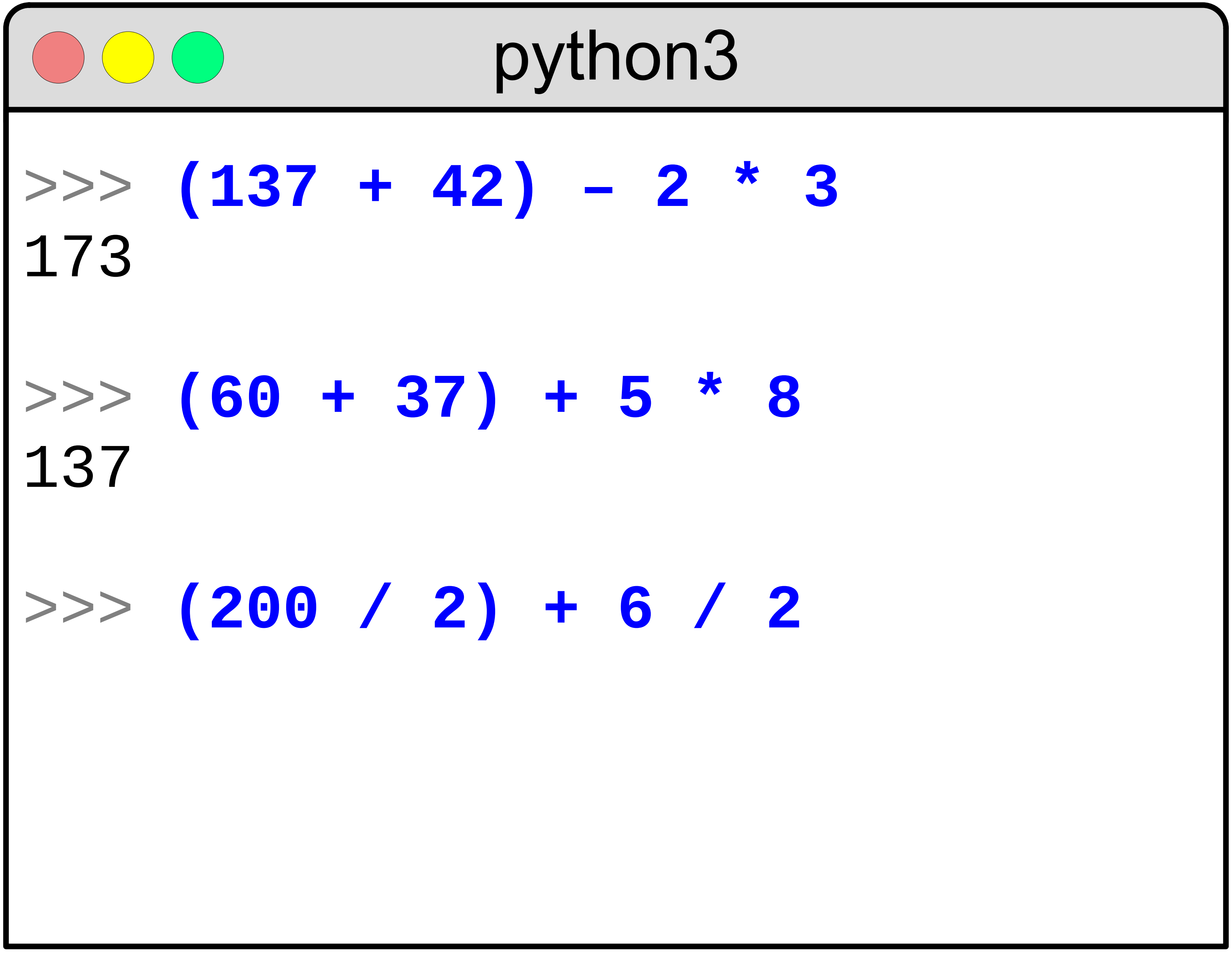
```
>>> (137 + 42) - 2 * 3
173

>>>
```

python3

```
>>> (137 + 42) - 2 * 3
173

>>> (60 + 37) + 5 * 8
```

python3

```
>>> (137 + 42) - 2 * 3
173

>>> (60 + 37) + 5 * 8
137

>>>
```


python3
>>> (137 + 42) - 2 * 3
173
>>> (60 + 37) + 5 * 8
137
>>> (200 / 2) + 6 / 2

python3

```
>>> (137 + 42) - 2 * 3
173

>>> (60 + 37) + 5 * 8
137

>>> (200 / 2) + 6 / 2
103.0

>>>
```

# Mad Libs for Arithmetic

| ( | ______ | ______ | ______ | ) | ______ | ______ | ______ | ______ |
|---|---|---|---|---|---|---|---|---|
| | **Int** | **Op** | **Int** | | **Op** | **Int** | **Op** | **Int** |

***Slide credit: Amy Liu***

# Mad Libs for Arithmetic

( 26 + 42 ) * 2 + 1

| Int | Op | Int | | Op | Int | Op | Int |
|---|---|---|---|---|---|---|---|
| 26 | + | 42 | ) | * | 2 | + | 1 |

*Slide credit: Amy Liu*

# Mad Libs for Arithmetic

( ______ ______ ______ ) ______ ______ ______ ______

Int Op Int Op Int Op Int

*Slide credit: Amy Liu*

# Mad Libs for Arithmetic

`( 7 * 5 ) / 5 - 49`

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| ( | 7 | * | 5 | ) | / | 5 | - | 49 |
| | **Int** | **Op** | **Int** | | **Op** | **Int** | **Op** | **Int** |

***Slide credit: Amy Liu***

# Mad Libs for Arithmetic

( ______ ______ ______ ) ______ ______ ______ ______

**Int** **Op** **Int** **Op** **Int** **Op** **Int**

This only lets us make arithmetic expressions of the form (**Int Op Int**) **Op Int Op Int**.

What about arithmetic expressions that don't follow this pattern?

*Slide credit: Amy Liu*

# Recursive Mad Libs

**Expr**

# Recursive Mad Libs

______

**Expr**

What can an arithmetic expression be?

# Recursive Mad Libs

`int`

**Expr**

What can an arithmetic expression be?

| | |
|---|---|
| `int` | A single number. |

# Recursive Mad Libs

______

**Expr**

What can an arithmetic expression be?

`int` A single number.

# Recursive Mad Libs

**Expr**

What can an arithmetic expression be?

| | |
|---|---|
| `int` | A single number. |
| **Expr Op Expr** | Two expressions joined by an operator. |

# Recursive Mad Libs

______ ______ ______

**Expr** **Op** **Expr**

What can an arithmetic expression be?

| | |
|---|---|
| `int` | A single number. |
| **Expr Op Expr** | Two expressions joined by an operator. |

# Recursive Mad Libs

**int**

_____ _____ _____

**Expr** **Op** **Expr**

What can an arithmetic expression be?

| | |
|---|---|
| **int** | A single number. |
| **Expr Op Expr** | Two expressions joined by an operator. |

# Recursive Mad Libs

int + ___
Expr Op Expr

What can an arithmetic expression be?

| | |
|---|---|
| `int` | A single number. |
| **Expr Op Expr** | Two expressions joined by an operator. |

# Recursive Mad Libs

int + ___

Expr Op Expr

What can an arithmetic expression be?

| | |
|---|---|
| `int` | A single number. |
| **Expr Op Expr** | Two expressions joined by an operator. |

# Recursive Mad Libs

| `int` | `+` | | | |
|---|---|---|---|---|
| **Expr** | **Op** | **Expr** | **Op** | **Expr** |

What can an arithmetic expression be?

| | |
|---|---|
| **`int`** | A single number. |
| **Expr Op Expr** | Two expressions joined by an operator. |

# Recursive Mad Libs

| int | + | | | |
|---|---|---|---|---|
| **Expr** | **Op** | **Expr** | **Op** | **Expr** |

What can an arithmetic expression be?

| | |
|---|---|
| `int` | A single number. |
| **Expr Op Expr** | Two expressions joined by an operator. |

# Recursive Mad Libs

| int | + | int | | |
|---|---|---|---|---|
| Expr | Op | Expr | Op | Expr |

What can an arithmetic expression be?

| | |
|---|---|
| int | A single number. |
| Expr Op Expr | Two expressions joined by an operator. |

# Recursive Mad Libs

| `int` | `+` | `int` | `×` | |
|---|---|---|---|---|
| **Expr** | **Op** | **Expr** | **Op** | **Expr** |

What can an arithmetic expression be?

| | |
|---|---|
| `int` | A single number. |
| **Expr Op Expr** | Two expressions joined by an operator. |

# Recursive Mad Libs

| `int` | `+` | `int` | `×` | `int` |
|---|---|---|---|---|
| **Expr** | **Op** | **Expr** | **Op** | **Expr** |

What can an arithmetic expression be?

| | |
|---|---|
| `int` | A single number. |
| **Expr Op Expr** | Two expressions joined by an operator. |

# Recursive Mad Libs

__________
**Expr**

What can an arithmetic expression be?

| | |
|---|---|
| `int` | A single number. |
| **Expr Op Expr** | Two expressions joined by an operator. |

# Recursive Mad Libs

**Expr**

What can an arithmetic expression be?

| | |
|---|---|
| `int` | A single number. |
| **Expr Op Expr** | Two expressions joined by an operator. |
| `(`**Expr**`)` | A parenthesized expression. |

# Recursive Mad Libs

( ______ )

**Expr**

What can an arithmetic expression be?

| | |
|---|---|
| `int` | A single number. |
| **Expr Op Expr** | Two expressions joined by an operator. |
| ( **Expr** ) | A parenthesized expression. |

# Recursive Mad Libs

( ______ )

**Expr**

What can an arithmetic expression be?

| | |
|---|---|
| `int` | A single number. |
| **Expr Op Expr** | Two expressions joined by an operator. |
| ( **Expr** ) | A parenthesized expression. |

# Recursive Mad Libs

( ______ ______ ______ )

**Expr** **Op** **Expr**

What can an arithmetic expression be?

| | |
|---|---|
| `int` | A single number. |
| **Expr Op Expr** | Two expressions joined by an operator. |
| `(`**Expr**`)` | A parenthesized expression. |

# Recursive Mad Libs

**(** ______ ______ ______ **)**

**Expr** **Op** **Expr**

What can an arithmetic expression be?

| | |
|---|---|
| `int` | A single number. |
| **Expr Op Expr** | Two expressions joined by an operator. |
| **(Expr)** | A parenthesized expression. |

# Recursive Mad Libs

**(int** _____ _____ _____ **)**

**Expr** **Op** **Expr**

What can an arithmetic expression be?

| | |
|---|---|
| `int` | A single number. |
| **Expr Op Expr** | Two expressions joined by an operator. |
| **(Expr)** | A parenthesized expression. |

# Recursive Mad Libs

(int / )

Expr Op Expr

What can an arithmetic expression be?

| | |
|---|---|
| `int` | A single number. |
| **Expr Op Expr** | Two expressions joined by an operator. |
| `(`**Expr**`)` | A parenthesized expression. |

# Recursive Mad Libs

`(int / ____ )`

Expr Op **Expr**

What can an arithmetic expression be?

| | |
|---|---|
| `int` | A single number. |
| **Expr Op Expr** | Two expressions joined by an operator. |
| `(`**Expr**`)` | A parenthesized expression. |

# Recursive Mad Libs

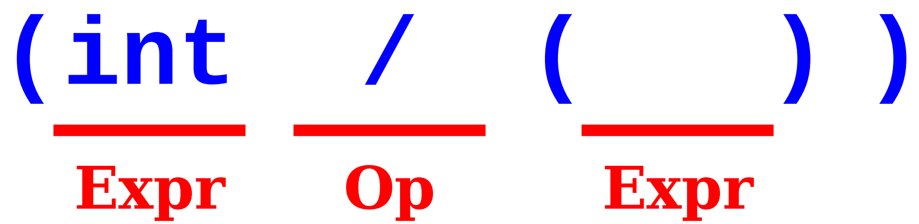
( int  /  ( ____ ) )

Expr  Op  Expr

What can an arithmetic expression be?

| | |
|---|---|
| `int` | A single number. |
| **Expr Op Expr** | Two expressions joined by an operator. |
| **(Expr)** | A parenthesized expression. |

# Recursive Mad Libs




What can an arithmetic expression be?

| | |
|---|---|
| `int` | A single number. |
| **Expr Op Expr** | Two expressions joined by an operator. |
| **(Expr)** | A parenthesized expression. |

# Recursive Mad Libs

`(int / ( ___ ))`

Expr Op Expr

What can an arithmetic expression be?

| | |
|---|---|
| `int` | A single number. |
| **Expr Op Expr** | Two expressions joined by an operator. |
| **(Expr)** | A parenthesized expression. |

# Recursive Mad Libs

`(int / (` ___ ___ ___ `))`

Expr   Op   Expr   Op   Expr

What can an arithmetic expression be?

| | |
|---|---|
| `int` | A single number. |
| **Expr Op Expr** | Two expressions joined by an operator. |
| `(`**Expr**`)` | A parenthesized expression. |

# Recursive Mad Libs

( int / ( ____ ____ ____ ) )

Expr Op Expr Op Expr

What can an arithmetic expression be?

| | |
|---|---|
| int | A single number. |
| Expr Op Expr | Two expressions joined by an operator. |
| (Expr) | A parenthesized expression. |

# Recursive Mad Libs


What can an arithmetic expression be?

| | |
|---|---|
| `int` | A single number. |
| **Expr Op Expr** | Two expressions joined by an operator. |
| **(Expr)** | A parenthesized expression. |

# Recursive Mad Libs

( int / ( int + ) )

Expr Op Expr Op Expr

What can an arithmetic expression be?

| | |
|---|---|
| `int` | A single number. |
| **Expr Op Expr** | Two expressions joined by an operator. |
| ( **Expr** ) | A parenthesized expression. |

# Recursive Mad Libs

**( int / ( int + int ) )**

| Expr | Op | Expr | Op | Expr |
|---|---|---|---|---|

What can an arithmetic expression be?

| | |
|---|---|
| **int** | A single number. |
| **Expr Op Expr** | Two expressions joined by an operator. |
| **( Expr )** | A parenthesized expression. |

A ***context-free grammar*** (or ***CFG***) is a recursive set of rules that define a language.

***(There's a bunch of specific requirements about what those rules can be; more on that in a bit.)***

# Arithmetic Expressions

- Here's how we might express the recursive rules from earlier as a CFG.

**Expr** → `int`

**Expr** → **Expr Op Expr**

**Expr** → `(`**Expr**`)`

**Op** → `+`

**Op** → `-`

**Op** → `×`

**Op** → `/`

This is called a ***production rule***. It says "if you see **Expr**, you can replace it with **Expr Op Expr**."

# Arithmetic Expressions

- Here's how we might express the recursive rules from earlier as a CFG.

**Expr** → `int`

**Expr** → **Expr Op Expr**

**Expr** → `(`**Expr**`)`

**Op** → `+`

**Op** → `-`

**Op** → `×`

**Op** → `/`

This one says "if you see **Op**, you can replace it with `-`."

# Arithmetic Expressions

- Here's how we might express the recursive rules from earlier as a CFG.

$$
\begin{aligned}
\mathbf{Expr} &\rightarrow \texttt{int} \\
\mathbf{Expr} &\rightarrow \mathbf{Expr}\ \mathbf{Op}\ \mathbf{Expr} \\
\mathbf{Expr} &\rightarrow \texttt{(}\mathbf{Expr}\texttt{)} \\
\mathbf{Op} &\rightarrow \texttt{+} \\
\mathbf{Op} &\rightarrow \texttt{-} \\
\mathbf{Op} &\rightarrow \times \\
\mathbf{Op} &\rightarrow \texttt{/}
\end{aligned}
$$

$$
\begin{aligned}
& \mathbf{Expr} \\
\Rightarrow\ & \mathbf{Expr}\ \mathbf{Op}\ \mathbf{Expr} \\
\Rightarrow\ & \mathbf{Expr}\ \mathbf{Op}\ \texttt{int} \\
\Rightarrow\ & \texttt{int}\ \mathbf{Op}\ \texttt{int} \\
\Rightarrow\ & \texttt{int}\ \texttt{/}\ \texttt{int}
\end{aligned}
$$

# Arithmetic Expressions

- Here's how we might express the recursive rules from earlier as a CFG.

**Expr** → `int`

**Expr** → **Expr Op Expr**

**Expr** → `(`**Expr**`)`

**Op** → `+`

**Op** → `-`

**Op** → `×`

**Op** → `/`

**Expr**
⇒ **Expr Op Expr**
⇒ **Expr Op** `int`
⇒ `int` **Op** `int`
⇒ `int / int`

These red symbols are called ***nonterminals***. They're placeholders that get expanded later on.

# Arithmetic Expressions

- Here's how we might express the recursive rules from earlier as a CFG.

**Expr** → `int`

**Expr** → **Expr Op Expr**

**Expr** → `(`**Expr**`)`

**Op** → `+`

**Op** → `-`

**Op** → `×`

**Op** → `/`

**Expr**

⇒ **Expr Op Expr**

⇒ **Expr Op** `int`

⇒ `int` **Op** `int`

⇒ `int / int`

The symbols in blue monospace are `terminals`. They're the final characters used in the string and never get replaced.

# Arithmetic Expressions

- Here's how we might express the recursive rules from earlier as a CFG.

**Expr** → `int`
**Expr** → **Expr Op Expr**
**Expr** → `(`**Expr**`)`
**Op** → `+`
**Op** → `-`
**Op** → `×`
**Op** → `/`

  **Expr**
⇒ **Expr Op Expr**
⇒ **Expr Op** `(`**Expr**`)`
⇒ **Expr Op** `(`**Expr Op Expr**`)`
⇒ **Expr** `×` `(`**Expr Op Expr**`)`
⇒ `int` `×` `(`**Expr Op Expr**`)`
⇒ `int` `×` `(int` **Op Expr**`)`
⇒ `int` `×` `(int` **Op** `int)`
⇒ `int` `×` `(int + int)`

# Context-Free Grammars

- Formally, a context-free grammar is a collection of four items:
  - a set of ***nonterminal symbols*** (also called ***variables***),
  - a set of ***terminal symbols*** (the ***alphabet*** of the CFG),
  - a set of ***production rules*** saying how each nonterminal can be replaced by a string of terminals and nonterminals, and
  - a ***start symbol*** (which must be a nonterminal) that begins the derivation. By convention, the start symbol is the one on the left-hand side of the first production.

**Expr** → `int`

**Expr** → **Expr Op Expr**

**Expr** → `(`**Expr**`)`

**Op** → `+`

**Op** → `-`

**Op** → `×`

**Op** → `/`

# Some CFG Notation

- In today's slides, capital letters in **Bold Red Uppercase** will represent nonterminals.
  - e.g. **A**, **B**, **C**, **D**
- Lowercase letters in **`blue monospace`** will represent terminals.
  - e.g. **`t`**, **`u`**, **`v`**, **`w`**
- Lowercase Greek letters in ***gray italics*** will represent arbitrary strings of terminals and nonterminals.
  - e.g. $\boldsymbol{\alpha}$, $\boldsymbol{\gamma}$, $\boldsymbol{\omega}$
- You don't need to use these conventions on your own; just make sure whatever you do is readable.

# A Notational Shorthand

**Expr** → `int`

**Expr** → **Expr Op Expr**

**Expr** → `(`**Expr**`)`

**Op** → `+`

**Op** → `-`

**Op** → `×`

**Op** → `/`

# A Notational Shorthand

**Expr** → `int` | **Expr Op Expr** | `(`**Expr**`)`

**Op** → `+` | `-` | `×` | `/`

# Derivations

**Expr**
⇒ **Expr Op Expr**
⇒ **Expr Op (Expr)**
⇒ **Expr Op (Expr Op Expr)**
⇒ **Expr × (Expr Op Expr)**
⇒ `int` × (**Expr Op Expr**)
⇒ `int` × (`int` **Op Expr**)
⇒ `int` × (`int` **Op** `int`)
⇒ `int` × (`int + int`)

- A sequence of zero or more steps where nonterminals are replaced by the right-hand side of a production is called a ***derivation***.
- If string $\alpha$ derives string $\omega$, we write $\alpha \Rightarrow^* \omega$.
- In the example on the left, we see that

**Expr** $\Rightarrow^*$ `int × (int + int)`.

**Expr** → `int` | **Expr Op Expr** | (**Expr**)
**Op** → `+` | `-` | `×` | `/`

# The Language of a Grammar

- If $G$ is a CFG with alphabet $\Sigma$ and start symbol **S**, then the ***language of G*** is the set

$$\mathscr{L}(G) = \{ \boldsymbol{\omega} \in \Sigma^* \mid \mathbf{S} \Rightarrow^* \boldsymbol{\omega} \}$$

- That is, $\mathscr{L}(G)$ is the set of strings of terminals derivable from the start symbol.

If $G$ is a CFG with alphabet $\Sigma$ and start symbol **S**, then the ***language of G*** is the set

$$\mathscr{L}(G) = \{ \omega \in \Sigma^* \mid \mathbf{S} \Rightarrow^* \omega \}$$

Consider the following CFG $G$ over $\Sigma = \{$**a**, **b**, **c**, **d**$\}$:

$$\mathbf{S} \rightarrow \mathbf{Sa} \mid \mathbf{dT}$$
$$\mathbf{T} \rightarrow \mathbf{bTb} \mid \mathbf{c}$$

Which of the following strings are in $\mathscr{L}(G)$?

**dca**
**dc**
**cad**
**bcb**
**dTaa**

***Answer at pollev.com/zhenglian740***

# Context-Free Languages

- A language $L$ is called a ***context-free language*** (or CFL) if there is a CFG $G$ such that $L = \mathscr{L}(G)$.
- Questions:
  - How are context-free and regular languages related?
  - How do we design context-free grammars for context-free languages?

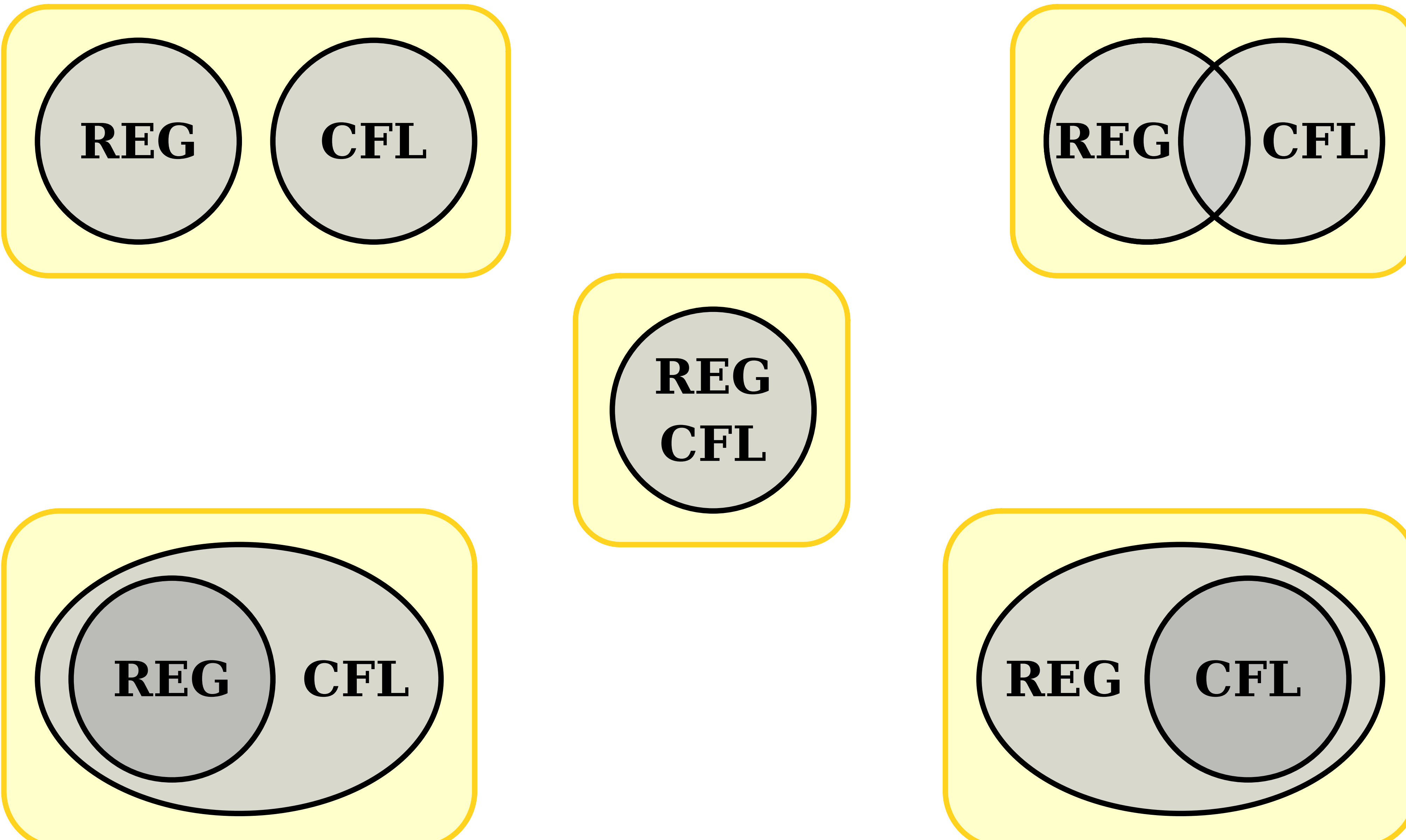
# Context-Free Languages

A language $L$ is called a ***context-free language*** (or CFL) if there is a CFG $G$ such that $L = \mathscr{L}(G)$.

Questions:

- How are context-free and regular languages related?
- How do we design context-free grammars for context-free languages?

# Five Possibilities



# CFGs and Regular Expressions

- CFGs consist purely of production rules of the form **A** → ***ω***. They do not have the regular expression operators * or ∪.
- You can use the symbols * and ∪ if you'd like in a CFG, but they just stand for themselves.
- Consider this CFG $G$:

$$\mathbf{S} \rightarrow \mathtt{a*b}$$

- Here, $\mathscr{L}(G) = \{\mathtt{a*b}\}$ and has cardinality one. That is, $\mathscr{L}(G) \neq \{\ \mathtt{a}^n\mathtt{b} \mid n \in \mathbb{N}\ \}$.

# CFGs and Regular Expressions

- ***Theorem:*** Every regular language is context-free.
- ***Proof idea:*** Show how to convert an arbitrary regular expression into a context-free grammar.

**a ( b ∪ ε ) c**

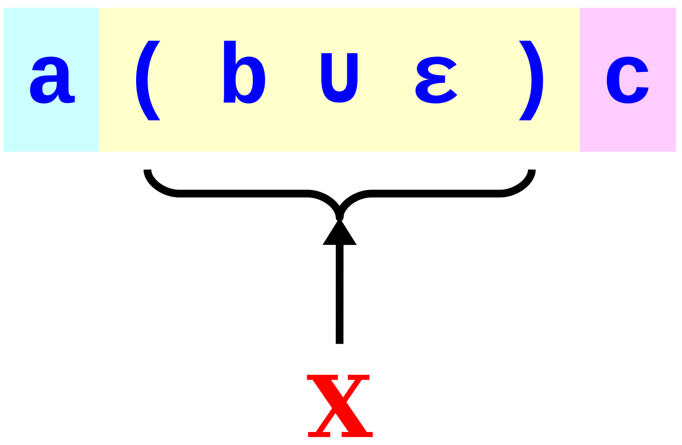
# CFGs and Regular Expressions

- ***Theorem:*** Every regular language is context-free.
- ***Proof idea:*** Show how to convert an arbitrary regular expression into a context-free grammar.

**a ( b ∪ ε ) c**

# CFGs and Regular Expressions

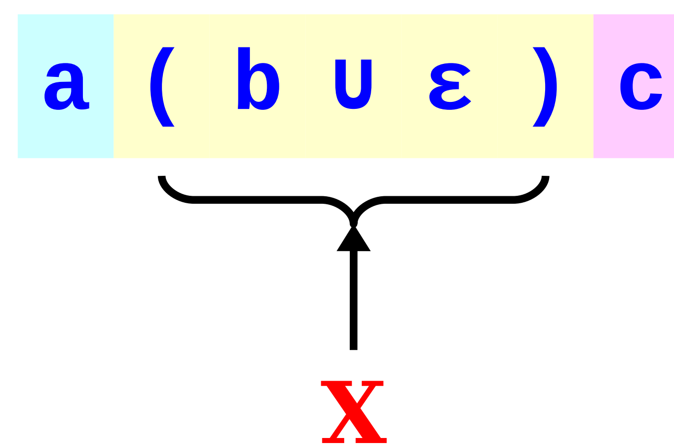
- ***Theorem:*** Every regular language is context-free.
- ***Proof idea:*** Show how to convert an arbitrary regular expression into a context-free grammar.



# CFGs and Regular Expressions

- ***Theorem:*** Every regular language is context-free.
- ***Proof idea:*** Show how to convert an arbitrary regular expression into a context-free grammar.

**S → aXc**

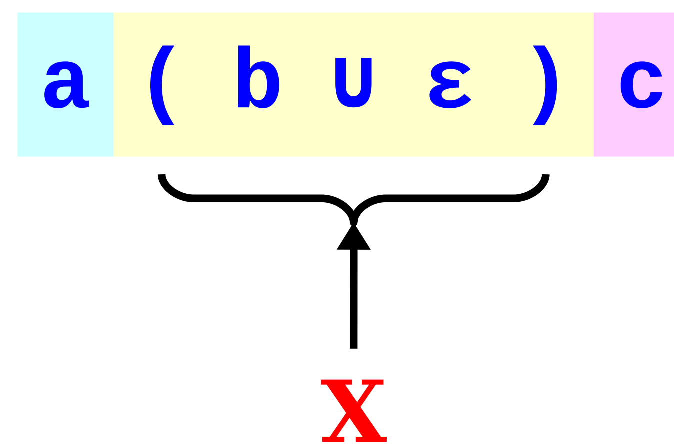


# CFGs and Regular Expressions

- ***Theorem:*** Every regular language is context-free.
- ***Proof idea:*** Show how to convert an arbitrary regular expression into a context-free grammar.

$S \rightarrow aXc$

$X \rightarrow b \mid \varepsilon$



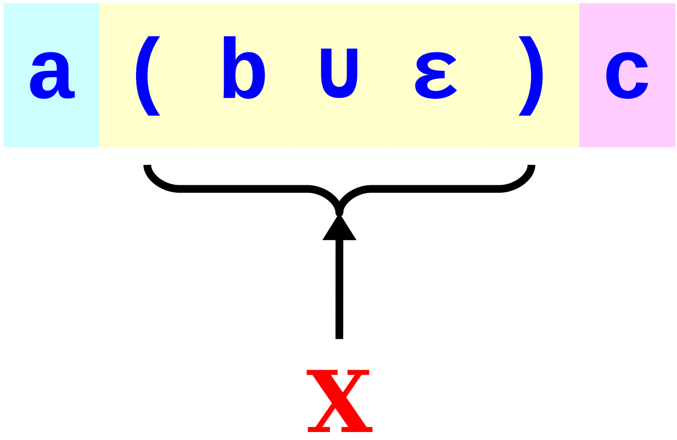
# CFGs and Regular Expressions

- ***Theorem:*** Every regular language is context-free.
- ***Proof idea:*** Show how to convert an arbitrary regular expression into a context-free grammar.

**S** → **aXc**

**X** → **b** | ε

It's totally fine for a production to replace a nonterminal with the empty string.



# CFGs and Regular Expressions

- ***Theorem:*** Every regular language is context-free.
- ***Proof idea:*** Show how to convert an arbitrary regular expression into a context-free grammar.

$(\mathbf{a} \cup \mathbf{b})^2 \mathbf{c}^*$

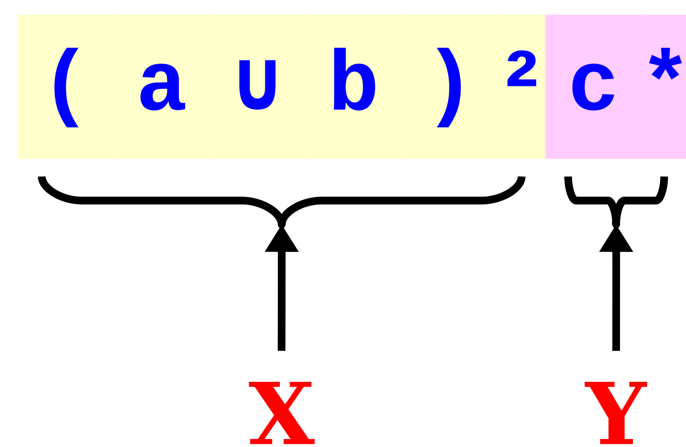
# CFGs and Regular Expressions

- ***Theorem:*** Every regular language is context-free.
- ***Proof idea:*** Show how to convert an arbitrary regular expression into a context-free grammar.

$(a \cup b)^2 c^*$

# CFGs and Regular Expressions

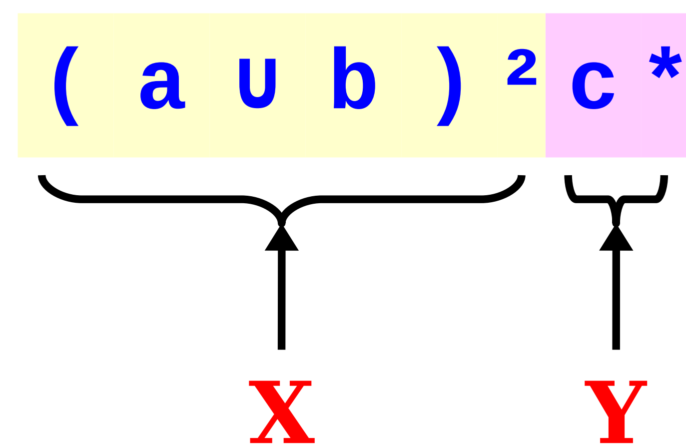
- ***Theorem:*** Every regular language is context-free.
- ***Proof idea:*** Show how to convert an arbitrary regular expression into a context-free grammar.



# CFGs and Regular Expressions

- ***Theorem:*** Every regular language is context-free.
- ***Proof idea:*** Show how to convert an arbitrary regular expression into a context-free grammar.

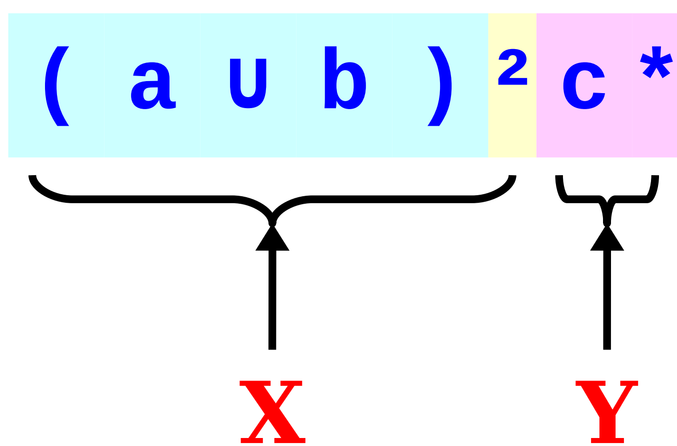
$S \rightarrow XY$



# CFGs and Regular Expressions

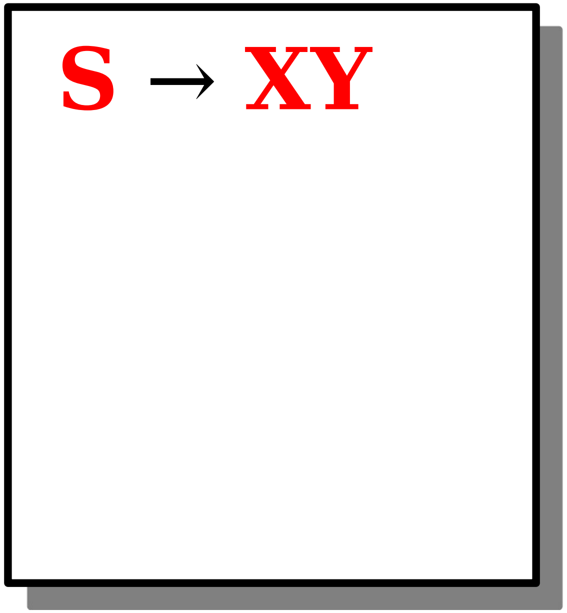
- ***Theorem:*** Every regular language is context-free.
- ***Proof idea:*** Show how to convert an arbitrary regular expression into a context-free grammar.

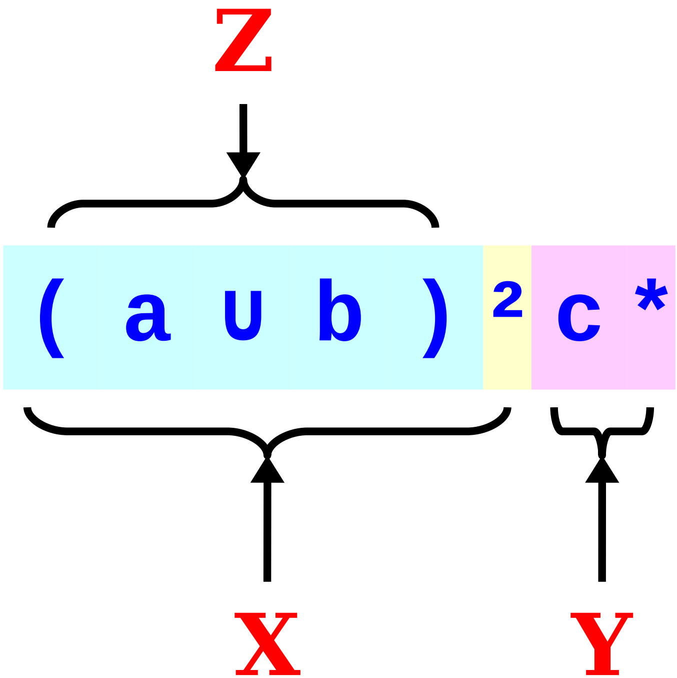
**S** → **XY**



# CFGs and Regular Expressions

- ***Theorem:*** Every regular language is context-free.
- ***Proof idea:*** Show how to convert an arbitrary regular expression into a context-free grammar.



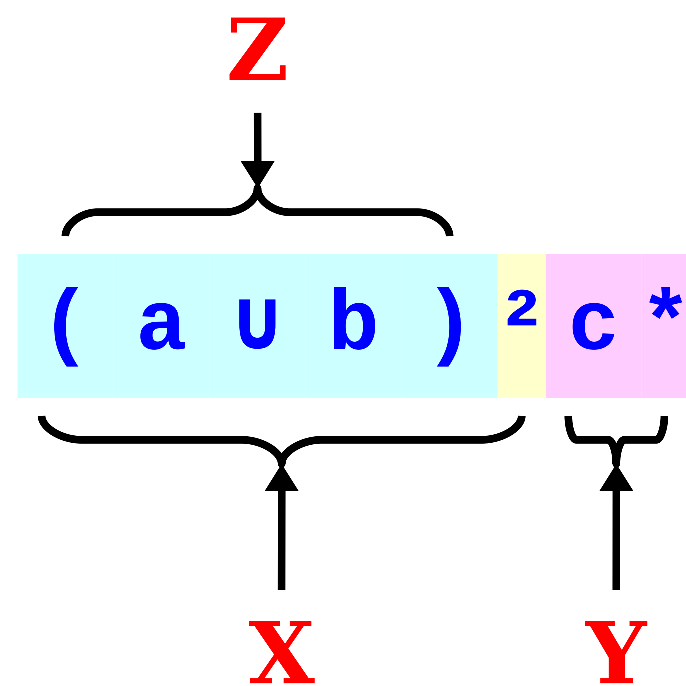



# CFGs and Regular Expressions

- ***Theorem:*** Every regular language is context-free.
- ***Proof idea:*** Show how to convert an arbitrary regular expression into a context-free grammar.

**S → XY**

**X → ZZ**



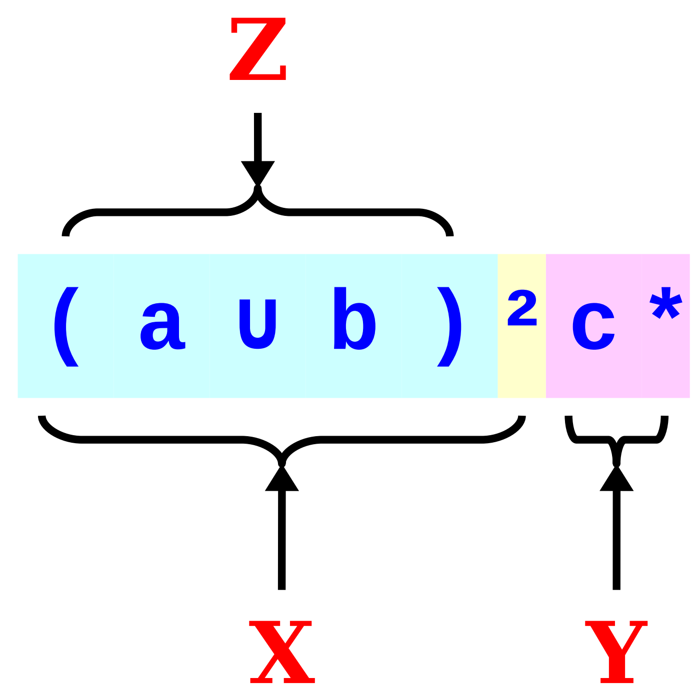
# CFGs and Regular Expressions

- ***Theorem:*** Every regular language is context-free.
- ***Proof idea:*** Show how to convert an arbitrary regular expression into a context-free grammar.

**S** → **XY**

**X** → **ZZ**

**Z** → **a** | **b**



# CFGs and Regular Expressions

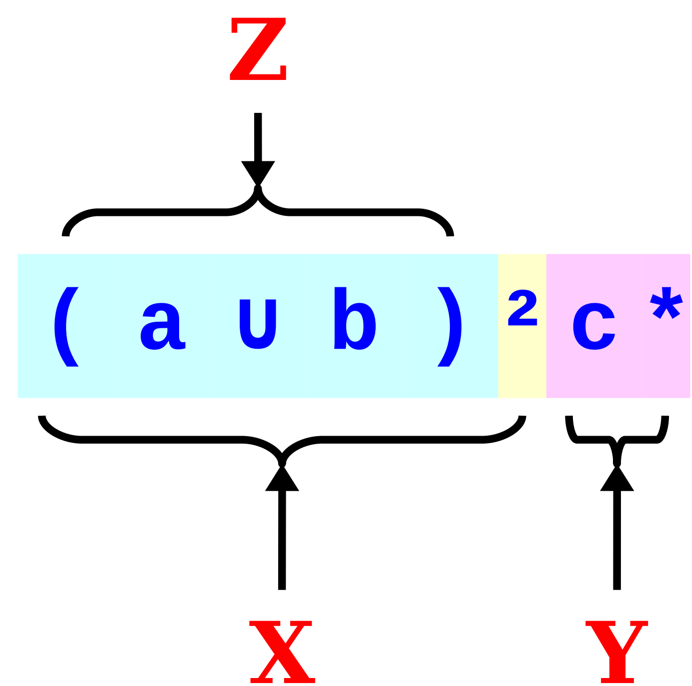
- ***Theorem:*** Every regular language is context-free.
- ***Proof idea:*** Show how to convert an arbitrary regular expression into a context-free grammar.

**S** → **XY**

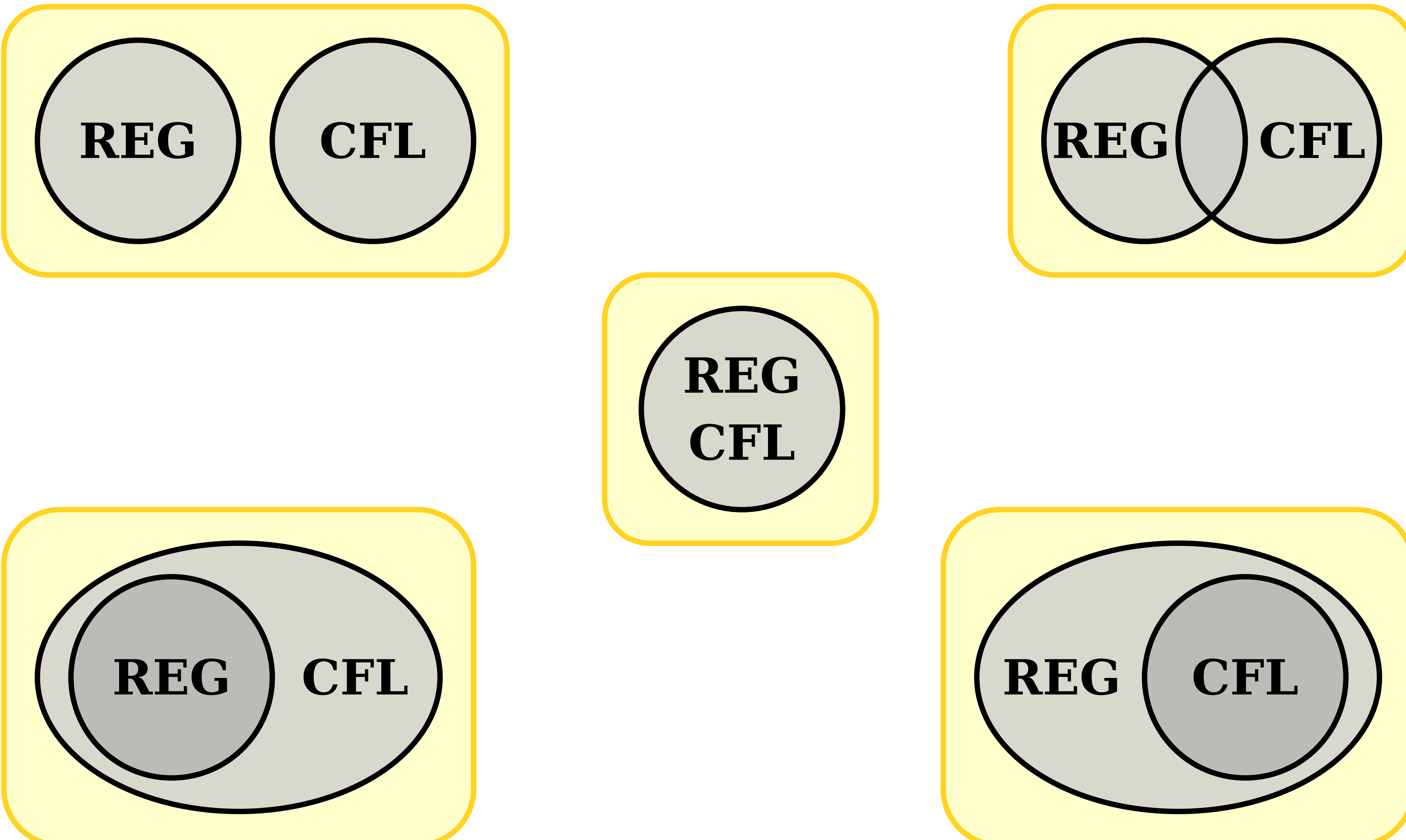
**X** → **ZZ**

**Z** → **a** | **b**

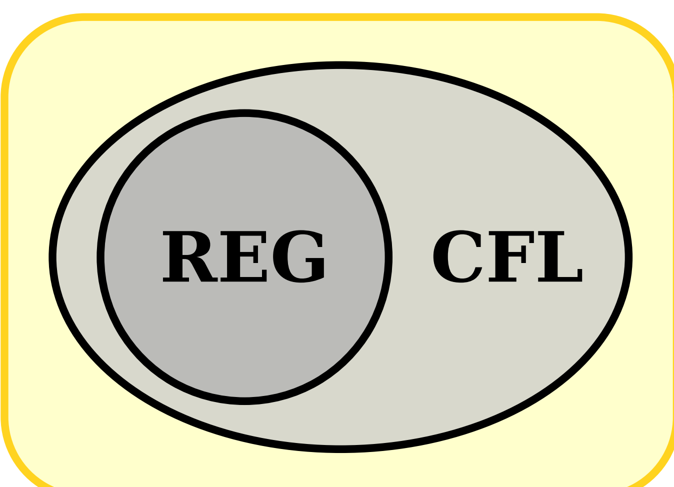
**Y** → **cY** | ε



# Two ~~Five~~ Possibilities



# Two ~~Five~~ Possibilities

# The Language of a Grammar

- Consider the following CFG *G*:

$$S \rightarrow aSb \mid \varepsilon$$

- What strings can this generate?

# The Language of a Grammar

- Consider the following CFG *G*:

$$S \rightarrow aSb \mid \varepsilon$$

- What strings can this generate?

S

# The Language of a Grammar

- Consider the following CFG *G*:

$$S \rightarrow aSb \mid \varepsilon$$

- What strings can this generate?

| | | |
|---|---|---|
| a | S | b |

# The Language of a Grammar

- Consider the following CFG *G*:

$$\mathbf{S} \rightarrow \mathtt{a}\mathbf{S}\mathtt{b} \mid \varepsilon$$

- What strings can this generate?

| a | S | b |
|---|---|---|

# The Language of a Grammar

- Consider the following CFG *G*:

$$S \rightarrow aSb \mid \varepsilon$$

- What strings can this generate?

| | | | | |
|---|---|---|---|---|
| a | a | S | b | b |

# The Language of a Grammar

- Consider the following CFG *G*:

$$\mathbf{S} \rightarrow \mathtt{a}\mathbf{S}\mathtt{b} \mid \varepsilon$$

- What strings can this generate?

| | | | | |
|---|---|---|---|---|
| **a** | **a** | **S** | **b** | **b** |

# The Language of a Grammar

- Consider the following CFG *G*:

$$\mathbf{S} \rightarrow \mathtt{a}\mathbf{S}\mathtt{b} \mid \varepsilon$$

- What strings can this generate?

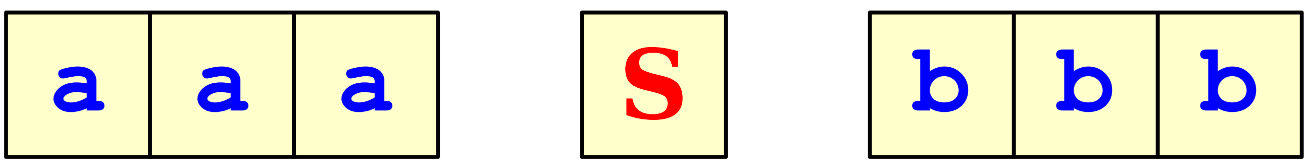
| | | | | | | |
|---|---|---|---|---|---|---|
| a | a | a | S | b | b | b |

# The Language of a Grammar

- Consider the following CFG *G*:

$$\mathbf{S} \rightarrow \mathtt{a}\mathbf{S}\mathtt{b} \mid \varepsilon$$

- What strings can this generate?



# The Language of a Grammar

- Consider the following CFG *G*:

$$S \rightarrow \mathtt{a}S\mathtt{b} \mid \varepsilon$$

- What strings can this generate?

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| a | a | a | a | S | b | b | b | b |

# The Language of a Grammar

- Consider the following CFG *G*:

$$\mathbf{S} \rightarrow \mathtt{a}\mathbf{S}\mathtt{b} \mid \varepsilon$$

- What strings can this generate?

| | | | |
|---|---|---|---|
| a | a | a | a |

| | | | |
|---|---|---|---|
| b | b | b | b |

# The Language of a Grammar

- Consider the following CFG *G*:

$$\mathbf{S} \rightarrow \mathtt{a}\mathbf{S}\mathtt{b} \mid \varepsilon$$

- What strings can this generate?

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| a | a | a | a | b | b | b | b |

# The Language of a Grammar

- Consider the following CFG $G$:

$$\mathbf{S} \rightarrow \mathtt{a}\mathbf{S}\mathtt{b} \mid \varepsilon$$

- What strings can this generate?

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| a | a | a | a | b | b | b | b |

$$\mathcal{L}(G) = \{ \mathtt{a}^n\mathtt{b}^n \mid n \in \mathbb{N} \}$$


Regular Languages
CFLs
All Languages

# Why the Extra Power?

- Why do CFGs have more power than regular expressions?
- ***Intuition:*** Derivations of strings have unbounded “memory.”

# Why the Extra Power?

- Why do CFGs have more power than regular expressions?
- ***Intuition:*** Derivations of strings have unbounded "memory."

$$\mathbf{S} \rightarrow \mathtt{a}\mathbf{S}\mathtt{b} \mid \varepsilon$$

# Why the Extra Power?

- Why do CFGs have more power than regular expressions?
- ***Intuition:*** Derivations of strings have unbounded "memory."

$$S \rightarrow \mathbf{a}S\mathbf{b} \mid \varepsilon$$

**S**

# Why the Extra Power?

- Why do CFGs have more power than regular expressions?
- ***Intuition:*** Derivations of strings have unbounded "memory."

**S → aSb | ε**

| | | |
|---|---|---|
| a | S | b |

# Why the Extra Power?

- Why do CFGs have more power than regular expressions?
- ***Intuition:*** Derivations of strings have unbounded "memory."

$$S \rightarrow aSb \mid \varepsilon$$

| a | S | b |
|---|---|---|

# Why the Extra Power?

- Why do CFGs have more power than regular expressions?
- ***Intuition:*** Derivations of strings have unbounded "memory."

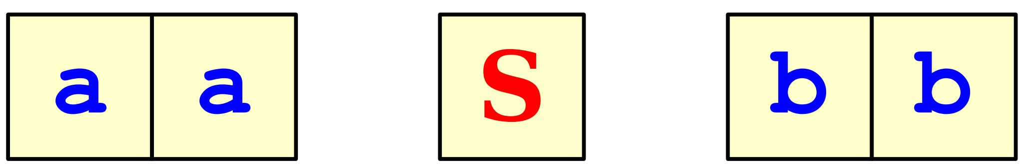
$$\mathbf{S} \rightarrow \mathtt{a}\mathbf{S}\mathtt{b} \mid \varepsilon$$

| | | | | |
|---|---|---|---|---|
| **a** | **a** | **S** | **b** | **b** |

# Why the Extra Power?

- Why do CFGs have more power than regular expressions?
- ***Intuition:*** Derivations of strings have unbounded "memory."

$$S \rightarrow aSb \mid \varepsilon$$



# Why the Extra Power?

- Why do CFGs have more power than regular expressions?
- ***Intuition:*** Derivations of strings have unbounded “memory.”

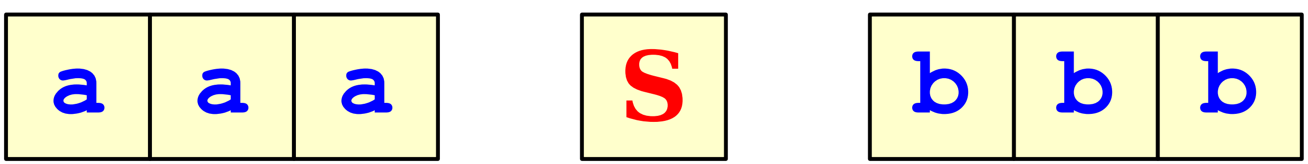
$$\mathbf{S} \rightarrow \mathtt{a}\mathbf{S}\mathtt{b} \mid \varepsilon$$

| | | | | | | |
|---|---|---|---|---|---|---|
| a | a | a | S | b | b | b |

# Why the Extra Power?

- Why do CFGs have more power than regular expressions?
- ***Intuition:*** Derivations of strings have unbounded "memory."

$$\mathbf{S} \rightarrow \mathtt{a}\mathbf{S}\mathtt{b} \mid \varepsilon$$



# Why the Extra Power?

- Why do CFGs have more power than regular expressions?
- ***Intuition:*** Derivations of strings have unbounded "memory."

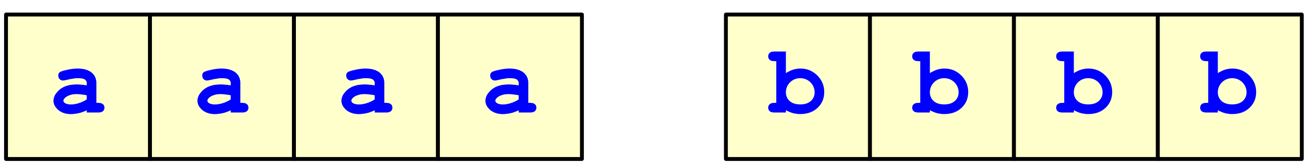
$$S \rightarrow \mathtt{a}S\mathtt{b} \mid \varepsilon$$

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| a | a | a | a | S | b | b | b | b |

# Why the Extra Power?

- Why do CFGs have more power than regular expressions?
- ***Intuition:*** Derivations of strings have unbounded “memory.”

$$S \rightarrow \mathtt{a}S\mathtt{b} \mid \varepsilon$$



# Why the Extra Power?

- Why do CFGs have more power than regular expressions?
- ***Intuition:*** Derivations of strings have unbounded "memory."

$$S \rightarrow aSb \mid \varepsilon$$

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| a | a | a | a | b | b | b | b |

Time-Out for Announcements!

# Problem Set Six

- Problem Set Five was due today at 5:30PM.
- Problem Set Six goes out today. It’s due next Friday at 5:30PM.
  - It’s all about regular expressions, properties of regular languages, and nonregular languages.

# Preparing for the Final Exam

- We've released two practice final exams. We strongly recommend sitting down and taking the practice exam under realistic exam conditions.
- There is also a gigantic compendium of CS103 practice problems on the course website.
  - You can search for problems based on the topics they cover, whether solutions are available, whether they're ones we particularly like, and whether they have solutions.
  - Please do ***not*** read the solutions to a problem until you have worked through it.

Back to CS103!

# Designing CFGs

- Like designing DFAs, NFAs, and regular expressions, designing CFGs is a craft.
- When thinking about CFGs:
  - ***Think recursively:*** Build up bigger structures from smaller ones.
  - ***Have a construction plan:*** Know in what order you will build up the string.
  - ***Store information in nonterminals:*** Have each nonterminal correspond to some useful piece of information.

# Designing CFGs

- Let $\Sigma = \{$**a**, **b**$\}$ and let $L = \{w \in \Sigma^* \mid w$ is a palindrome $\}$
- We can design a CFG for $L$ by thinking inductively:
  - Base case: ε, **a**, and **b** are palindromes.
  - If ***ω*** is a palindrome, then **a*ω*a** and **b*ω*b** are palindromes.
  - No other strings are palindromes.

**S → ε | a | b | aSa | bSb**

# Designing CFGs

- Let $\Sigma = \{$**{**, **}**$\}$ and let $L = \{w \in \Sigma^* \mid w$ is a string of balanced braces $\}$
- Some sample strings in $L$:

**{{{}}}**

**{{}}{}**

**{{}{}}{{}{}}**

**{{{{{}}}{{}}}}**

**ε**

**{}{}**

# Designing CFGs

- Let Σ = {**{**, **}**} and let $L$ = {$w$ ∈ Σ* | $w$ is a string of balanced braces }
- Let's think about this recursively.
  - Base case: the empty string is a string of balanced braces.
  - Recursive step: Look at the closing brace that matches the first open brace.

**{{{}{{}}}{{}}}{{}}{{{}}}**

# Designing CFGs

- Let $\Sigma = \{ \{, \} \}$ and let $L = \{ w \in \Sigma^* \mid w$ is a string of balanced braces $\}$
- Let's think about this recursively.
  - Base case: the empty string is a string of balanced braces.
  - Recursive step: Look at the closing brace that matches the first open brace.

{{{}{{}}}{{}}}{{}}{{{}}}

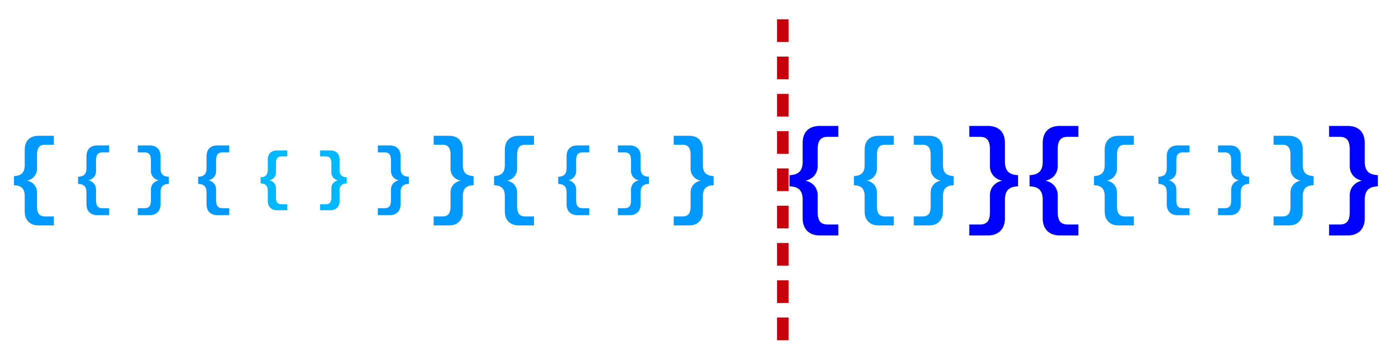
# Designing CFGs

- Let Σ = {**{**, **}**} and let $L$ = {$w$ ∈ Σ* | $w$ is a string of balanced braces }
- Let's think about this recursively.
  - Base case: the empty string is a string of balanced braces.
  - Recursive step: Look at the closing brace that matches the first open brace.

**{{{}{{}}}{{}}}{{}}{{{}}}**

# Designing CFGs

- Let Σ = {{, }} and let $L$ = {$w$ ∈ Σ* | $w$ is a string of balanced braces }
- Let's think about this recursively.
  - Base case: the empty string is a string of balanced braces.
  - Recursive step: Look at the closing brace that matches the first open brace.



# Designing CFGs

- Let $\Sigma = \{$ **{**, **}** $\}$ and let $L = \{ w \in \Sigma^* \mid w$ is a string of balanced braces $\}$
- Let's think about this recursively.
  - Base case: the empty string is a string of balanced braces.
  - Recursive step: Look at the closing brace that matches the first open brace. Removing the first brace and the matching brace forms two new strings of balanced braces.

$$\mathbf{S} \rightarrow \mathbf{\{S\}S} \mid \boldsymbol{\varepsilon}$$

# Designing CFGs

- Let $\Sigma = \{$**a**, **b**$\}$ and let $L = \{ w \in \Sigma^* \mid w$ has the same number of **a**'s and **b**'s $\}$

How many of the following CFGs have language $L$?

**S → aSb | bSa | ε**

**S → abS | baS | ε**

**S → abSba | baSab | ε**

**S → SbaS | SabS | ε**

***Answer at pollev.com/cs103***

# Designing CFGs

- Let $\Sigma = \{$**a**, **b**$\}$ and let $L = \{w \in \Sigma^* \mid w$ has the same number of **a**'s and **b**'s $\}$

How many of the following CFGs have language $L$?

**S** → **aSb** | **bSa** | **ε**

**S** → **abS** | **baS** | **ε**

**S** → **abSba** | **baSab** | **ε**

**S** → **SbaS** | **SabS** | **ε**

# Designing CFGs

- Let Σ = {**a**, **b**} and let $L$ = {$w$ ∈ Σ* | $w$ has the same number of **a**'s and **b**'s }

How many of the following CFGs have language $L$?

**S** → **aSb** | **bSa** | **ε**

**S** → **abS** | **baS** | **ε**

**S** → **abSba** | **baSab** | **ε**

**S** → **SbaS** | **SabS** | **ε**

# Designing CFGs

- Let $\Sigma$ = {**a**, **b**} and let $L$ = { $w \in \Sigma^*$ | $w$ has the same number of **a**'s and **b**'s }

How many of the following CFGs have language $L$?

**S → aSb | bSa | ε**

**S → abS | baS | ε**

**S → abSba | baSab | ε**

**S → SbaS | SabS | ε**

# Designing CFGs

- Let $\Sigma = \{$**a**, **b**$\}$ and let $L = \{w \in \Sigma^* \mid w$ has the same number of **a**'s and **b**'s $\}$

How many of the following CFGs have language $L$?

**S** → **aSb** | **bSa** | **ε**

**S** → **abS** | **baS** | **ε**

**S** → **abSba** | **baSab** | **ε**

**S** → **SbaS** | **SabS** | **ε**

# Designing CFGs: A Caveat

- When designing a CFG for a language, make sure that it
  - generates all the strings in the language and
  - never generates a string outside the language.
- The first of these can be tricky – make sure to test your grammars!
- You'll design your own CFG for this language on Problem Set 8.

# CFG Caveats II

- Is the following grammar a CFG for the language { $\mathbf{a}^n\mathbf{b}^n \mid n \in \mathbb{N}$ }?

$$\mathbf{S} \rightarrow \mathbf{aSb}$$

- What strings in {**a**, **b**}* can you derive?
  - Answer: ***None!***
- What is the language of the grammar?
  - Answer: **Ø**
- When designing CFGs, make sure your recursion actually terminates!

# Designing CFGs

- When designing CFGs, remember that each nonterminal can be expanded out independently of the others.
- Let $\Sigma = \{\mathbf{a}, \stackrel{?}{=}\}$ and let $L = \{\mathbf{a}^n \stackrel{?}{=} \mathbf{a}^n \mid n \in \mathbb{N}\}$.
- Is the following a CFG for $L$?

$$\mathbf{S} \to \mathbf{X} \stackrel{?}{=} \mathbf{X}$$

$$\mathbf{X} \to \mathbf{aX} \mid \boldsymbol{\varepsilon}$$

$$\begin{aligned} & \mathbf{S} \\ \Rightarrow\ & \mathbf{X} \stackrel{?}{=} \mathbf{X} \\ \Rightarrow\ & \mathbf{aX} \stackrel{?}{=} \mathbf{X} \\ \Rightarrow\ & \mathbf{aaX} \stackrel{?}{=} \mathbf{X} \\ \Rightarrow\ & \mathbf{aa} \stackrel{?}{=} \mathbf{X} \\ \Rightarrow\ & \mathbf{aa} \stackrel{?}{=} \mathbf{aX} \\ \Rightarrow\ & \mathbf{aa} \stackrel{?}{=} \mathbf{a} \end{aligned}$$

# Finding a Build Order

- Let $\Sigma = \{$**a**, **≟**$\}$ and let $L = \{$ $\mathbf{a}^n$**≟**$\mathbf{a}^n \mid n \in \mathbb{N}$ $\}$.
- To build a CFG for $L$, we need to be more clever with how we construct the string.
  - If we build the strings of **a**'s independently of one another, then we can't enforce that they have the same length.
  - ***Idea:*** Build both strings of **a**'s at the same time.
- Here's one possible grammar based on that idea:

**S** → **≟** | **aSa**

**S**
⇒ **aSa**
⇒ **aaSaa**
⇒ **aaaSaaa**
⇒ **aaa≟aaa**

# Storing Information in Nonterminals

- ***Key idea:*** Different non-terminals should represent different states or different types of strings.
  - For example, different phases of the build, or different possible structures for the string.
  - Think like the same ideas from DFA/NFA design where states in your automata represent pieces of information.

# Storing Information in Nonterminals

- Let $\Sigma = \{$**a**, **b**$\}$ and let $L = \{ w \in \Sigma^* \mid |w| \equiv_3 0$ and all the characters in the first third of $w$ are the same $\}$.
- Examples:

| | |
|---|---|
| **ε** $\in L$ | **a** $\notin L$ |
| **abb** $\in L$ | **b** $\notin L$ |
| **bab** $\in L$ | **ababab** $\notin L$ |
| **aababa** $\in L$ | **aabaaaaaa** $\notin L$ |
| **bbbbbb** $\in L$ | **bbbb** $\notin L$ |

# Storing Information in Nonterminals

- Let $\Sigma = \{$**a**, **b**$\}$ and let $L = \{ w \in \Sigma^* \mid |w| \equiv_3 0$ and all the characters in the first third of $w$ are the same $\}$.
- Examples:

| | |
|---|---|
| **ε** $\in L$ | **a** $\notin L$ |
| **a** ⁞ **bb** $\in L$ | **b** $\notin L$ |
| **b** ⁞ **ab** $\in L$ | **ab** ⁞ **abab** $\notin L$ |
| **aa** ⁞ **baba** $\in L$ | **aab** ⁞ **aaaaaa** $\notin L$ |
| **bb** ⁞ **bbbb** $\in L$ | **bbbb** $\notin L$ |

# Storing Information in Nonterminals

- Let $\Sigma = \{$**a**, **b**$\}$ and let $L = \{w \in \Sigma^* \mid |w| \equiv_3 0$ and all the characters in the first third of $w$ are the same $\}$.

# Storing Information in Nonterminals

- Let $\Sigma = \{$**a**, **b**$\}$ and let $L = \{ w \in \Sigma^* \mid |w| \equiv_3 0$ and all the characters in the first third of $w$ are the same $\}$.
- One approach:

| | |
|---|---|
| **aaa** | **bab** |
| **abb** | **bbb** |
| **aaabab** | **bbabbb** |
| **aababa** | **bbbaaaaaa** |
| **aaaaaaaaa** | **bbbbbabaa** |

***Observation 1:***

Strings in this language are either: the first third is **a**s or the first third is **b**s.

# Storing Information in Nonterminals

- Let $\Sigma = \{$**a**, **b**$\}$ and let $L = \{ w \in \Sigma^* \mid |w| \equiv_3 0$ and all the characters in the first third of $w$ are the same $\}$.
- One approach:

| | |
|---|---|
| **aaa** | **bab** |
| **abb** | **bbb** |
| **aaabab** | **bbabbb** |
| **aababa** | **bbbaaaaaa** |
| **aaaaaaaaa** | **bbbbbabaa** |

# Storing Information in Nonterminals

- Let $\Sigma = \{$**a**, **b**$\}$ and let $L = \{ w \in \Sigma^* \mid |w| \equiv_3 0$ and all the characters in the first third of $w$ are the same $\}$.
- One approach:

| | |
|---|---|
| **aaa** | **bab** |
| **abb** | **bbb** |
| **aaabab** | **bbabbb** |
| **aababa** | **bbbaaaaaa** |
| **aaaaaaaaa** | **bbbbbabaa** |

***Observation 2:***

Amongst these strings, for every **a** I have in the first third, I need two other characters in the last two thirds.

# Storing Information in Nonterminals

- Let $\Sigma = \{$**a**, **b**$\}$ and let $L = \{ w \in \Sigma^* \mid |w| \equiv_3 0$ and all the characters in the first third of $w$ are the same $\}$.
- One approach:

**aaa** bab

**abb** bbb

**aaabab** bbabbb

aaaaaa

bbabaa

***Observation 2:***

Amongst these strings, for every **a** I have in the first third, I need two other characters in the last two thirds.

This pattern of "for every x I see here, I need a y somewhere else in the string" is very common in CFGs!

# Storing Information in Nonterminals

- Let $\Sigma = \{$**a**, **b**$\}$ and let $L = \{ w \in \Sigma^* \mid |w| \equiv_3 0$ and all the characters in the first third of $w$ are the same $\}$.
- One approach:

| | | |
|---|---|---|
| **aaa** | **bab** | ***Observation 2:*** |
| **abb** | **bbb** | Amongst these strings, for every **a** I have in the first third, I need two other characters in the last two thirds. |
| **aaabab** | **bbabbb** | |
| **aababa** | **bbbaaaaaa** | |
| **aaaaaaaaa** | **bbbbbabaa** | |

**A** → **aAXX** | **ε**    **X** → **a** | **b**

# Storing Information in Nonterminals

- Let $\Sigma = \{$**a**, **b**$\}$ and let $L = \{ w \in \Sigma^* \mid |w| \equiv_3 0$ and all the characters in the first third of $w$ are the same $\}$.
- One approach:

**aaa** bab

**abb**

**aaabab**

**aababa**

**aaaaaaaaa** bbbbbabaa

Here the nonterminal **A** represents "a string where the first third is **a**'s" and the nonterminal **X** represents "any character"

**A** → **aAXX** | **ε** **X** → **a** | **b**

# Storing Information in Nonterminals

- Let $\Sigma = \{$**a**, **b**$\}$ and let $L = \{ w \in \Sigma^* \mid |w| \equiv_3 0$ and all the characters in the first third of $w$ are the same $\}$.
- One approach:

| | |
|---|---|
| **aaa** | bab |
| **abb** | bbb |
| **aaabab** | bbabbb |
| **aababa** | bbbaaaaaa |
| **aaaaaaaaa** | bbbbbabaa |

**A** → **aAXX** | **ε**   **X** → **a** | **b**

# Storing Information in Nonterminals

- Let $\Sigma = \{$**a**, **b**$\}$ and let $L = \{ w \in \Sigma^* \mid |w| \equiv_3 0$ and all the characters in the first third of $w$ are the same $\}$.
- One approach:

| | |
|---|---|
| **aaa** | **bab** |
| **abb** | **bbb** |
| **aaabab** | **bbabbb** |
| **aababa** | **bbbaaaaaa** |
| **aaaaaaaaa** | **bbbbbabaa** |

**B** → **bBXX** | **ε**     **X** → **a** | **b**

# Storing Information in Nonterminals

- Let $\Sigma = \{\mathbf{a}, \mathbf{b}\}$ and let $L = \{w \in \Sigma^* \mid |w| \equiv_3 0$ and all the characters in the first third of $w$ are the same $\}$.
- Tying everything together:

**S** → **A** | **B**

**A** → **aAXX** | **ε**

**B** → **bBXX** | **ε**

**X** → **a** | **b**

# Storing Information in Nonterminals

- Let $\Sigma = \{$**a**, **b**$\}$ and let $L = \{ w \in \Sigma^* \mid |w| \equiv_3 0$ and all the characters in the first third of $w$ are the same $\}$.
- Tying everything together:

**S** → **A** | **B**

**A** → **a**AXX | ε

**B** → **b**BXX | ε

**X** → **a** | **b**

Overall strings in this language either follow the pattern of **A** or **B**.

# Storing Information in Nonterminals

- Let $\Sigma = \{$**a**, **b**$\}$ and let $L = \{w \in \Sigma^* \mid |w| \equiv_3 0$ and all the characters in the first third of $w$ are the same $\}$.
- Tying everything together:

**S** → **A** | **B**

**A** → **aAXX** | **ε**

**B** → **bBXX** | **ε**

**X** → **a** | **b**

**A** represents "strings where the first third is **a**'s"

# Storing Information in Nonterminals

- Let $\Sigma = \{$**a**, **b**$\}$ and let $L = \{ w \in \Sigma^* \mid |w| \equiv_3 0$ and all the characters in the first third of $w$ are the same $\}$.
- Tying everything together:

**S** → **A** | **B**

**A** → **aAXX** | **ε**

**B** → **bBXX** | **ε**

**X** → **a** | **b**

**B** represents "strings where the first third is **b**'s"

# Storing Information in Nonterminals

- Let $\Sigma = \{\mathbf{a}, \mathbf{b}\}$ and let $L = \{w \in \Sigma^* \mid |w| \equiv_3 0$ and all the characters in the first third of $w$ are the same $\}$.
- Tying everything together:

$$S \rightarrow A \mid B$$

$$A \rightarrow \mathbf{a}AXX \mid \varepsilon$$

$$B \rightarrow \mathbf{b}BXX \mid \varepsilon$$

$$X \rightarrow \mathbf{a} \mid \mathbf{b}$$

X represents "either an **a** or a **b**"

# Function Prototypes

- Let $\Sigma$ = {**`void`**, **`int`**, **`double`**, **`name`**, **`(`**, **`)`**, **`,`**, **`;`**}.
- Let's write a CFG for C-style function prototypes!
- Examples:
  - `void name(int name, double name);`
  - `int name();`
  - `int name(double name);`
  - `int name(int, int name, int);`
  - `void name(void);`

# Function Prototypes

- Here's one possible grammar:
  - **S** → **Ret** `name` `(`**Args**`);`
  - **Ret** → **Type** | `void`
  - **Type** → `int` | `double`
  - **Args** → `ε` | `void` | **ArgList**
  - **ArgList** → **OneArg** | **ArgList**, **OneArg**
  - **OneArg** → **Type** | **Type** `name`
- Fun question to think about: what changes would you need to make to support pointer types?

# Summary of CFG Design Tips

- Look for recursive structures where they exist: they can help guide you toward a solution.
- Keep the build order in mind – often, you'll build two totally different parts of the string concurrently.
  - Usually, those parts are built in opposite directions: one's built left-to-right, the other right-to-left.
- Use different nonterminals to represent different structures.

# Applications of Context-Free Grammars

# CFGs for Programming Languages

```
BLOCK  →  STMT
       |  { STMTS }

STMTS  →  ε
       |  STMT STMTS

STMT   →  EXPR;
       |  if (EXPR) BLOCK
       |  while (EXPR) BLOCK
       |  do BLOCK while (EXPR);
       |  BLOCK
       |  …

EXPR   →  identifier
       |  constant
       |  EXPR + EXPR
       |  EXPR – EXPR
       |  EXPR * EXPR
       |  …
```

# Grammars in Compilers

- One of the key steps in a compiler is figuring out what a program "means."
- This is usually done by defining a grammar showing the high-level structure of a programming language.
- There are certain classes of grammars (LL(1) grammars, LR(1) grammars, LALR(1) grammars, etc.) for which it's easy to figure out how a particular string was derived.
- Tools like `yacc` or `bison` automatically generate parsers from these grammars.
- Curious to learn more? ***Take CS143!***

# Natural Language Processing

- By building context-free grammars for actual languages and applying statistical inference, it's possible for a computer to recover the likely meaning of a sentence.
  - In fact, CFGs were first called ***phrase-structure grammars*** and were introduced by Noam Chomsky in his seminal work *Syntactic Structures*.
  - They were then adapted for use in the context of programming languages, where they were called ***Backus-Naur forms***.
- The **Stanford Parser** project is one place to look for an example of this.
- Want to learn more? Take CS124 or CS224N!

# Next Time

- ***Turing Machines***
  - What does a computer with unbounded memory look like?
  - How would you program it?
  - What can you do with it?